

JAVA JPA

@MOUSSA CAMARA

La persistance des données avec JPA

Littéralement « **Java Persistence API** », il s'agit d'un standard faisant partie intégrante de la plate-forme Java EE, une spécification qui définit un ensemble de règles permettant la gestion de la correspondance entre des objets Java et une base de données, ou autrement formulé la gestion de la persistance.

Il nous permet:

- De manipuler assez facilement les entités de notre application grâce aux annotations @
- De créer des relations(associations) entre différentes tables (ORM) grâce un fournisseur tel que Hibernate

Les entités

Une entité JPA est, par définition, une classe Java qui doit avoir les propriétés suivantes :

- Elle doit posséder un constructeur vide, public ou protected. Rappelons que ce constructeur vide existe par défaut si aucun constructeur n'existe dans la classe. Dans le cas contraire, il doit être ajouté explicitement.
- Elle ne doit pas être final, et aucune de ses méthodes ne peut être final.
- Une entité JPA ne peut pas être une interface ou une énumération.
- Une entité JPA peut être une classe concrète ou abstraite.

Une entité est signalée par l'annotation **@Entity** sur la classe.

De plus, une entité JPA doit disposer d'un ou plusieurs attributs définissant un identifiant grâce à l'annotation @Id.

Qu'est ce que Hibernate ?

Hibernate est un framework open source gérant la persistance des objets en base de données relationnelle.

Hibernate est adaptable en termes d'architecture, il peut donc être utilisé aussi bien dans un développement client lourd, que dans un environnement web léger de type Apache Tomcat ou dans un environnement Java EE complet : WebSphere, JBoss Application Server et Oracle WebLogic Server.



Installation Hibernate

Nous allons installer Hibernate:

<http://hibernate.org/orm/releases/>

On crée un dossier lib dans notre projet

Une fois téléchargée, on dézippe et copie tous les fichiers .jar présents dans lib/required dans lib.

Pour la connection à mysql, on allons télécharger le jar directement: <https://dev.mysql.com/downloads/connector/j/>

Connector/J 8.0.22

Select Operating System:

Platform Independent

[Looking for previous GA versions?](#)

Platform Independent (Architecture Independent), Compressed TAR Archive (mysql-connector-java-8.0.22.tar.gz)	8.0.22	3.8M	Download
MD5: eb4e915366543844a80a06ea111ec6f7 Signature			
Platform Independent (Architecture Independent), ZIP Archive (mysql-connector-java-8.0.22.zip)	8.0.22	4.5M	Download
MD5: 2aa66a62a2f3330730ec77b1c025646f Signature			



We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.

Tester la connection à la BDD

Tous les .jar sont déposés dans le dossier. On va builder notre projet:

Clique droit sur le nom du projet / aller propriétés/ Java Build path/ libraries/adds Jars/ tonProjet/lib et choisir le jar de mysql.

Enfin créer une class main pour tester la connection

NB: NE PAS OUBLIER DE FAIRE DE MEME POUR LES JAR DE HIBERNATE:

Selectionner tous les jars depuis le dossier lib et clique droit / build path / add build path

```
main.java
1 package testjdbc;
2
3 import java.sql.Connection;
4 import java.sql.DriverManager;
5 import java.sql.SQLException;
6
7 public class main {
8
9     public static void main(String[] args) {
10         String url = "jdbc:mysql://localhost:3306/test";
11         String username = "root";
12         String password = "";
13
14         System.out.println("Connecting database...");
15
16         try (Connection connection = DriverManager.getConnection(url, username, password)) {
17             System.out.println("Database connected!");
18         } catch (SQLException e) {
19             throw new IllegalStateException("Cannot connect the database!", e);
20         }
21     }
22 }
23
24
```

Persistance

Maintenant qu'on a bien installé notre hibernate et jdbc pour mysql, on va devoir établir une connection entre notre code source à la base de donnée. Nous allons créer un xml avec les accès à notre base de donnée.

On crée (s'il n'existe pas) un dossier META-INF dans src. On va importer les persistences avec les propriétés d'accès à la base de donnée ainsi que le driver

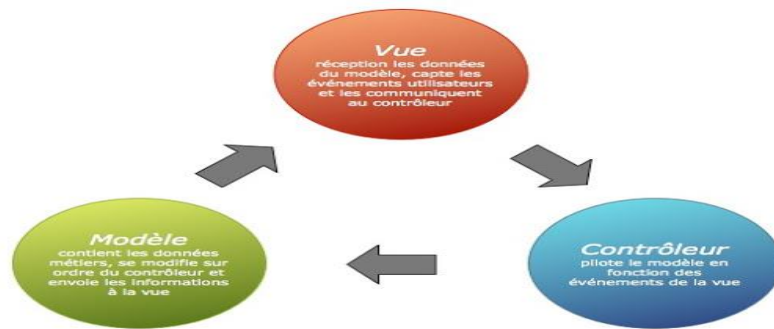
Persistence

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="WebStore">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>fr.koor.webstore.business.Article</class>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost/WebStore" />
      <property name="javax.persistence.jdbc.user" value="#user#" />
      <property name="javax.persistence.jdbc.password" value="#password#" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
    </properties>
  </persistence-unit>
</persistence>
```

JPA - Java Persistence API



MVC



Le pattern MVC est une façon d'organiser un code source autour de trois piliers:

- Le Model qui représente la structure des données. Leur définition ainsi que les fonctions qui leurs sont propres et qu'elles peuvent avoir.
- La View (ou les vues) représente l'interface graphique à livrer au client qui en fait la requête. Avoir le code lié à l'interface isolé de la logique métier ou des données permet de faire des modifications à l'interface graphique sans avoir à se soucier de casser du code métier ou la structure des données.
- Le(s) Controller(s) sont au coeur de la logique métier de votre application. Ils se situent entre les vues et le model.

MVC - Le modèle

Nous allons créer un modèle de donnée

- Article
 - id
 - titre
 - resume

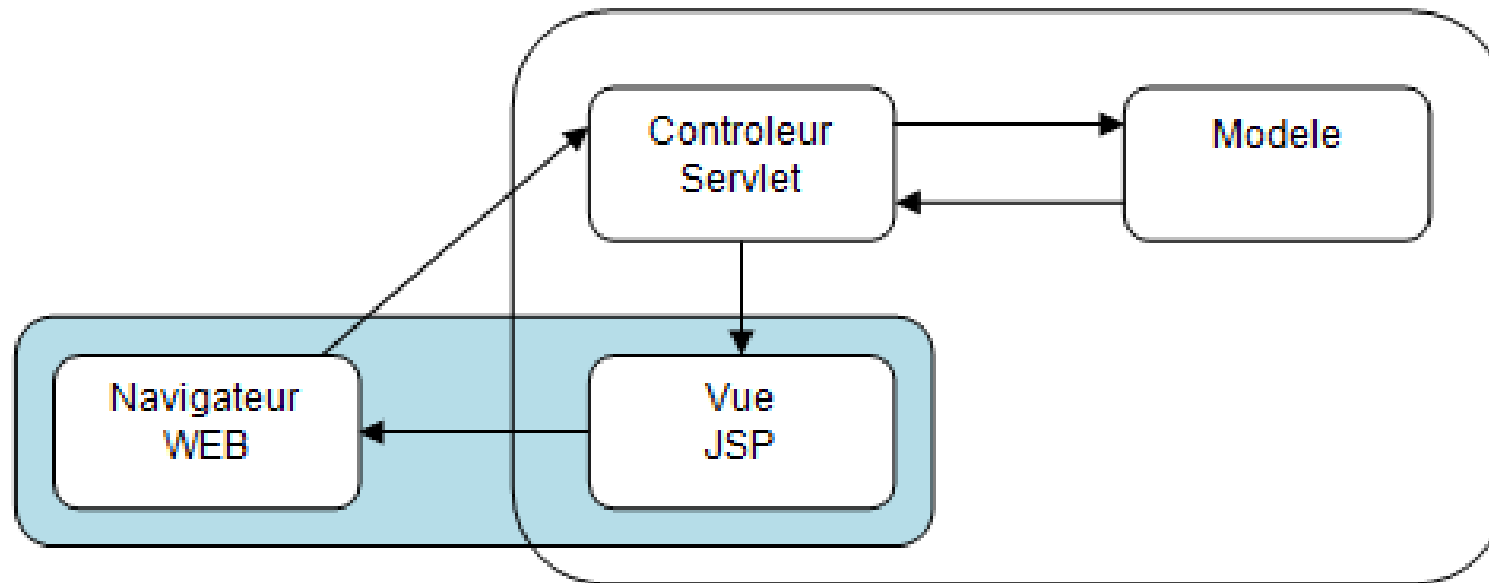
NB: Dans toute entité, il faut faire un constructeur vide

```
@Entity @Table(name="Article")
public class Article {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String titre;
    private String resume;
    public Article() {
        super();
    }
    public String getTitre() {
        return titre;
    }
    public void setTitre(String titre) {
        this.titre = titre;
    }
    public String getResume() {
        return resume;
    }
    public void setResume(String resume) {
        this.resume = resume;
    }
}
```

MVC - La vue - view

Ici, nos vues (en jsp) seront dans le dossier WebContent



Le controller

Dans notre controller, nous allons préparer notre connection à la base de donnée.

- **EntityManagerFactory emf:** permettant de manipuler des entités JPA. Il modélise notre unité de persistance. C'est à partir de lui que l'on peut construire des objets de type EntityManager, qui nous permettent d'interagir avec la base.
- **EntityManager em:** propose des fonctionnalités pour les manipuler (ajout, modification suppression, recherche). Ce gestionnaire est responsable de la gestion de l'état des entités et de leur persistance dans la base de données.
- **emf = Persistence.createEntityManagerFactory("moussa");** Cette instruction crée une fabrique d'objets de type EntityManagerFactory capable de fournir des objets EntityManager destinés à gérer des contextes de persistance liés à l'unité de persistance nommée **moussa**.
- **em = emf.createEntityManager();**

EntityManager

À partir d'une instance d'EntityManager, nous allons pouvoir manipuler les entités afin de les créer, les modifier, les charger ou les supprimer.

Les méthodes:

- **Find:** *permet de rechercher une entité en donnant sa clé primaire. Un appel à cette méthode ne déclenche pas forcément une requête SELECT vers la base de données.*
- **persist**
- **Merge:** permet de réaliser les UPDATE des entités en base de données.
- **Detach:** détache une entité, c'est-à-dire que l'instance passée en paramètre ne sera plus gérée par l'EntityManager. Ainsi, lors du commit de la transaction, les modifications faites sur l'entité détachée ne seront pas prises en compte.
- **Refresh:** annule toutes les modifications faites sur l'entité durant la transaction courante et recharge son état à partir des valeurs en base de données.
- **Remove:** supprime une entité

Exemple d'un article

```
package persistenceJpa;

import java.util.List;

public class ArticleController {

    EntityManagerFactory emf;
    EntityManager em;

    public ArticleController() {
        emf = Persistence.createEntityManagerFactory("moussa");

        em = emf.createEntityManager();
    }

    public void create(Article article) {
        em.getTransaction().begin();
        em.merge(article);
        em.getTransaction().commit();
        emf.close();
    }

    public void listArticle() {
        List<Article> articles = em.createQuery( "from Article", Article.class ).getResultList();
        for (Article article : articles) {
            System.out.println(" Titre: " + article.getTitre() );
            System.out.println(" Résumé: " + article.getResume() );
        }
    }
}
```

CRUD – CREATE

```
public void create(Article article) {
```

```
    // Ouvrir ou démarrer une transaction
```

```
    em.getTransaction().begin();
```

```
    // Une opération MERGE sur une entité persistante attache cette entité à l' entity manager courant. On utilise cette opération pour  
    associer une entité à un autre entity manager que celui qui a été utilisé pour la créer ou la lire.
```

```
    em.merge(article);
```

```
    // Commit permet de valider la transaction qu'on peut aussi annuler par le rollback
```

```
    em.getTransaction().commit();
```

```
    //Close permet de liberer les ressources
```

```
    emf.close();
```

```
}
```

CRUD - Read

```
public void listArticle() {
```

```
    // Récupération de toutes les données de la table Article
```

```
    List<Article> articles = em.createQuery( "from Article", Article.class ).getResultList();
```

```
    ◦ // Parcourir pour afficher chaque element du tableau
```

```
    for (Article article : articles) {
```

```
        System.out.println(" Titre: " + article.getTitre() );
```

```
        System.out.println(" Résumé: " + article.getResume() );
```

```
    }
```

```
}
```


CRUD - Update / Edit

```
public void editArticle(Article article, int id) {  
    em.getTransaction().begin();  
  
    article = em.find(Article.class, id);  
    article.setTitre("modif");  
    article.setResume("modifRésumé");  
  
    em.getTransaction().commit();  
  
    emf.close();  
}
```

CRUD - DELETE

```
public void remove(Article article, int id) {  
    em.getTransaction().begin();  
    article = em.find(Article.class, id);  
  
    em.remove(article);  
    em.getTransaction().commit();  
    em.close();  
}
```

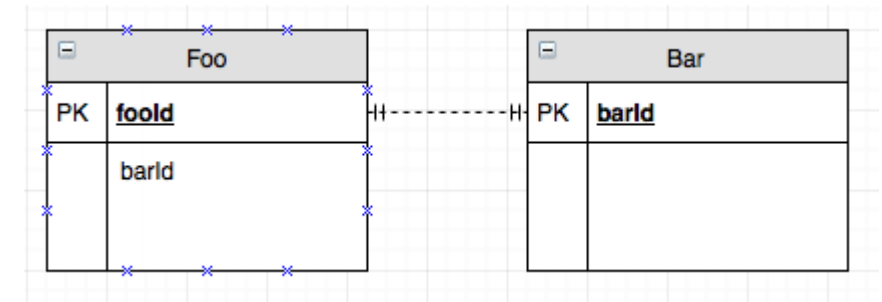
Association Mapping

- Many-To-One, Unidirectionnel
- One-To-One, Unidirectionnel
- One-To-One, Bidirectionnel
- One-To-Many, Unidirectionnel avec une table de jointure
- Many-To-Many, Unidirectionnel
- Many-To-Many, Bidirectionnel

Les directions

Unidirectionnel (d'un seul côté)

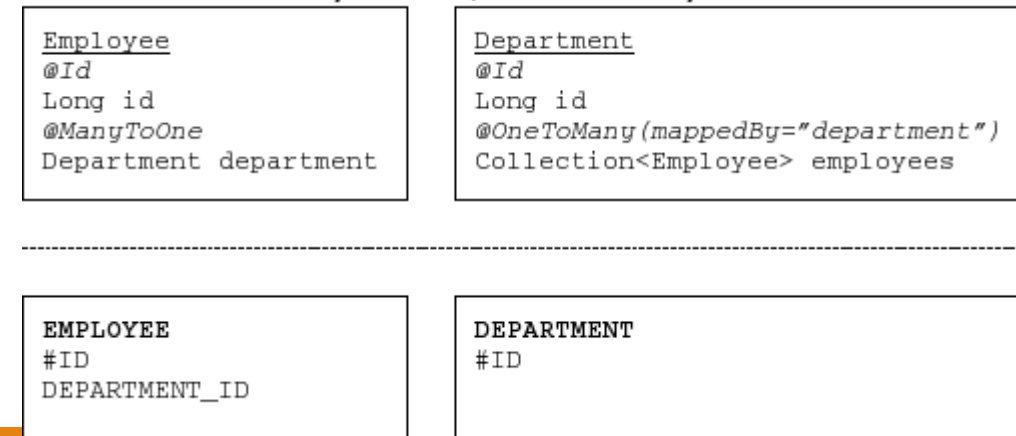
Cette table de jointure porte une clé étrangère vers la clé primaire de la première table, et une clé étrangère vers la clé primaire de la deuxième table.



Bidirectionnel

Parfois, il est nécessaire de construire un lien entre deux objets qui soit navigables dans les deux sens. D'un point de vue objet, cela signifie que chaque objet dispose d'un attribut pointant sur l'autre instance.

Bidirectional ManyToOne/OneToMany



Exemple - @OneToOne

Supposons qu'un etudiant a un et un seul numero d'étudiant ou adresse mail

Students table

Student ID	<input type="text" value="12345"/>
Last Name	<input type="text" value="Tang"/>
First Name	<input type="text" value="Sophie"/>

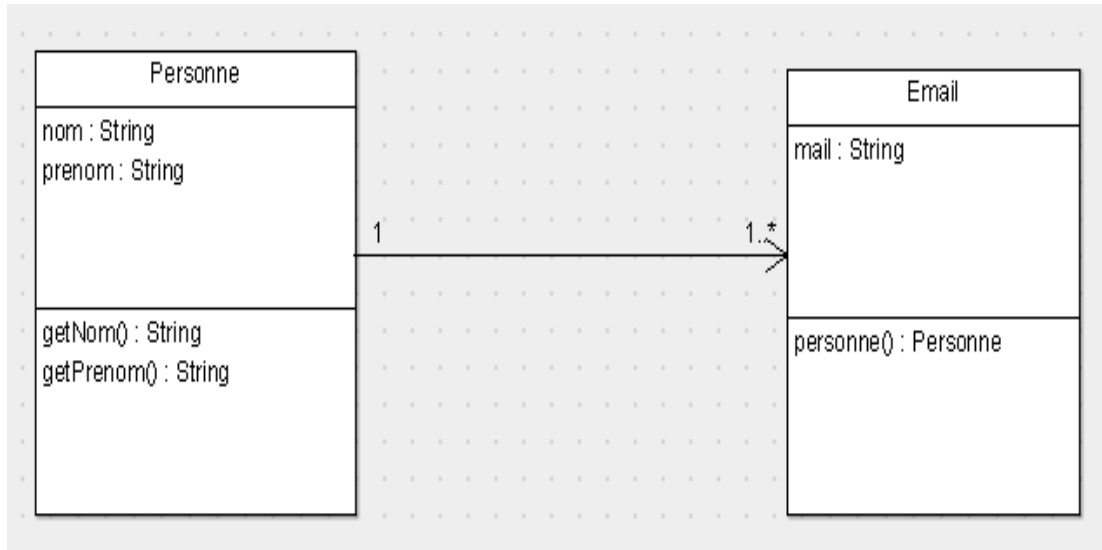
Contact Info table

Student ID	<input type="text" value="12345"/>
City	<input type="text" value="New York"/>
Phone	<input type="text" value="408-555-3456"/>



Exemple - @OneToMany

Supposons que nous avons une classe personne qui peut avoir plusieurs emails.



@OneToMany

```
public class Prospect extends Persistent {  
    ...  
    @OneToMany(mappedBy = "prospect")  
    private Set<ProspectComment> prospectComments;  
}
```

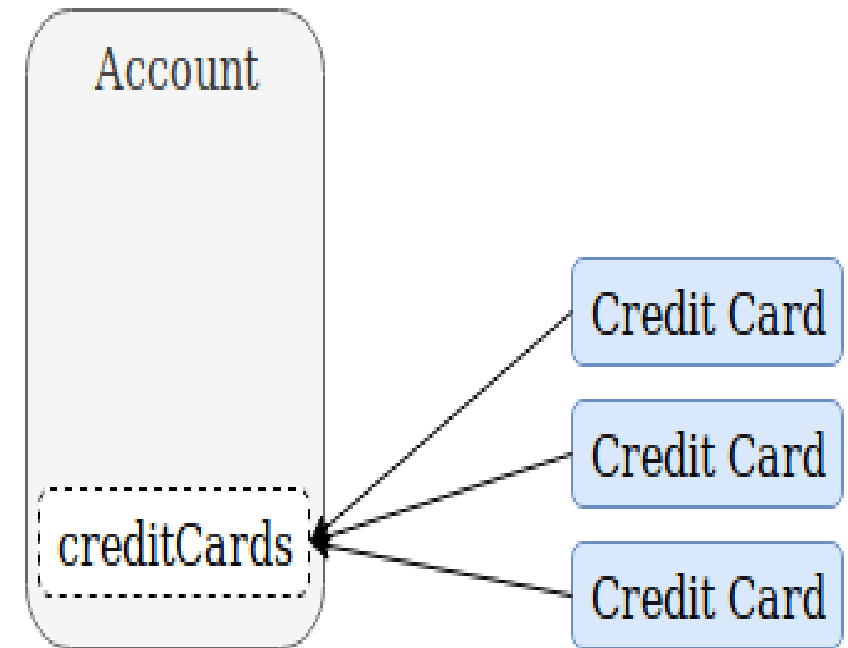
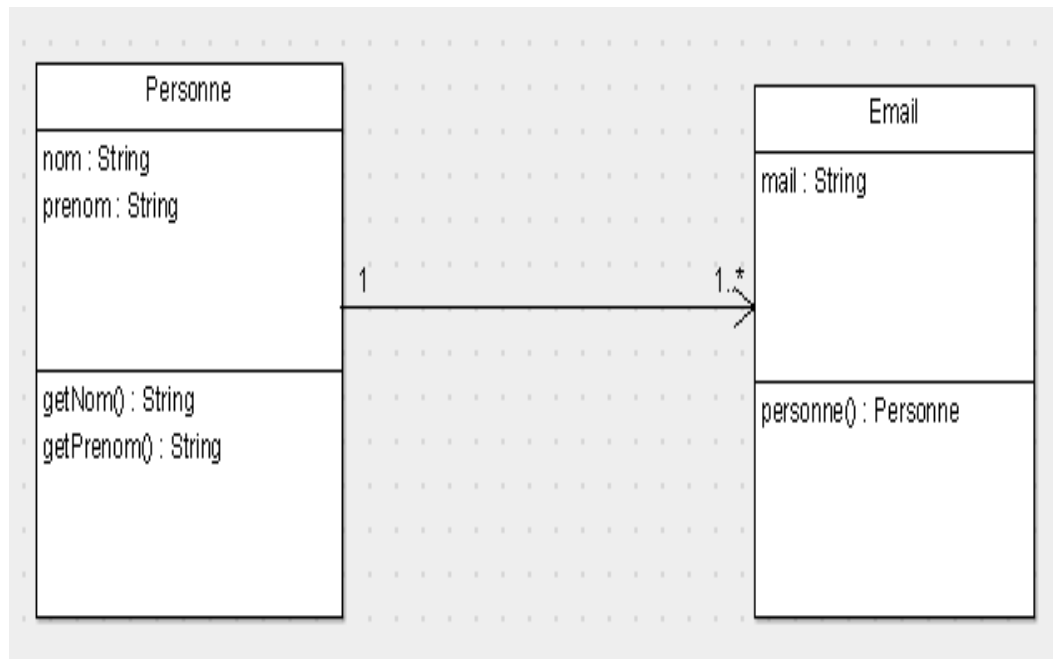
```
@Entity  
public class ProspectComment extends Persistent {  
    ...  
    @ManyToOne()  
    @JoinColumn(name = "PROSPECT_ID")  
    private Prospect prospect;  
}
```

MAUI.PROSPECT	
PROSPECT_ID	NUMBER(18) PK
...	...

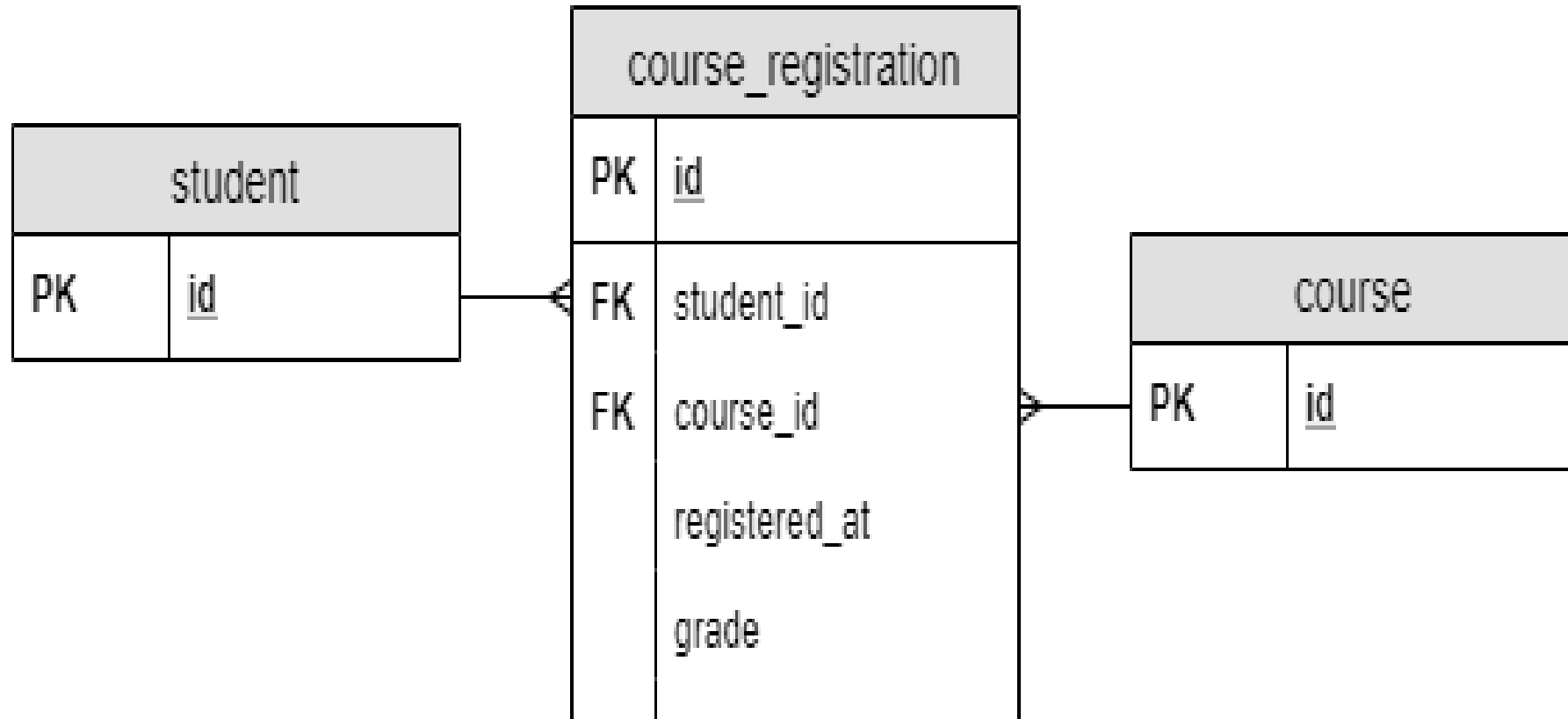
MAUI.PRSP_COMMENT	
PRSP_COMMENT_ID	NUMBER(18) PK
PROSPECT_ID	NUMBER(18) FK NOT NULL
...	...

Exemple - @ManyToOne

A l'inverse, nous pouvons avoir plusieurs emails pour une personne ou un compte avec plusieurs carte de crédit



Exemple - @ManyToMany



Exemple de ManyToMany

Pour cet exemple, nous allons supposer qu'un utilisateur peut avoir plusieurs roles inversement.

Dans notre class User:

```
@ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "personnes_roles",
        joinColumns={ @JoinColumn(name = "personne") },
        inverseJoinColumns = { @JoinColumn(name = "role") })
    private List<Role> roles;
```

Ici on va créer une table `personnes_roles` qui contiendra les id personne et rôle

Dans notre classe Rôle:

```
@ManyToMany(mappedBy = "roles")  
    private List<Personne> users;
```

Ici on va « mapper » les rôles aux utilisateurs.

Exemple ManyToOne

On avait tout à l'heure un OneToMany dans la classe personne on va maintenant faire l'inverse c'est à dire un ManyToOne depuis la classe Email.

Cela donne:

@ManyToOne

@JoinColumn(name="personne")

private Personne **personne**;

Au passage au spring boot

SPRING BOOT



SPRING
Framework

FRAMEWORK

Un Framework est une boite à outils pour un développeur web.

Frame signifie cadre et work se traduit par travail. Un Framework contient des composants autonomes qui permettent de faciliter le développement d'un site web ou d'une application.

Ces composants résolvent des problèmes souvent rencontrés par les développeurs (CRUD, arborescence, normes, sécurités, etc.). Ils permettent donc de gagner du temps lors du développement du site.

Histoire Spring

Spring est un framework open source pour construire et définir l'infrastructure d'une application Java3, dont il facilite le développement et les tests.

En 2004, Rod Johnson a écrit le livre Expert One-on-One J2EE Design and Development4 qui explique les raisons de la création de Spring.

« Spring est effectivement un conteneur dit « léger », c'est-à-dire une infrastructure similaire à un serveur d'applications J2EE. Il prend donc en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Le gros avantage par rapport aux serveurs d'application est qu'avec Spring, les classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications J2EE et des EJBs). C'est en ce sens que Spring est qualifié de conteneur « léger ». »

Spring Boot

Spring s'appuie principalement sur l'intégration de trois concepts clés :

- L'inversion de contrôle est assurée de deux façons différentes : la recherche de dépendances et l'injection de dépendances ;
- La programmation orientée aspect ;
- Une couche d'abstraction.

La couche d'abstraction permet d'intégrer d'autres frameworks et bibliothèques avec une plus grande facilité. Cela se fait par l'apport ou non de couches d'abstraction spécifiques à des frameworks particuliers. Il est ainsi possible d'intégrer un module d'envoi de mails plus facilement.

L'inversion de contrôle :

1. La recherche de dépendance : consiste pour un objet à interroger le conteneur, afin de trouver ses dépendances avec les autres objets. C'est un cas de fonctionnement similaire aux EJBs ;
2. L'injection de dépendances : cette injection peut être effectuée de trois manières possibles :
 - l'injection de dépendance via le constructeur,
 - l'injection de dépendance via les modificateurs (setters),
 - l'injection de dépendance via une interface.

Spring: les modules

Spring propose aussi un ensemble très complet de modules additionnels qui ne cesse de s'enrichir pour faciliter la mise en œuvre de certaines fonctionnalités dans les applications.

On retrouve des Framework tels que: Spring IO Plateform, Spring boot, Spring Cloud,



**Spring
Boot**



Spring Cloud

<https://spring.io/>

Spring Boot

Lorsqu'on concevait des applications avec Spring Framework par exemple on perdait beaucoup de temps pour gérer la configuration qui se faisait en xml 🤔

XML = Extensible Markup Language ou « langage de balisage extensible »

Avec Spring Boot, cette configuration est faite ou presque pour nous

Avec SpringBoot, nous avons déjà quelques modules installés et paramétrés

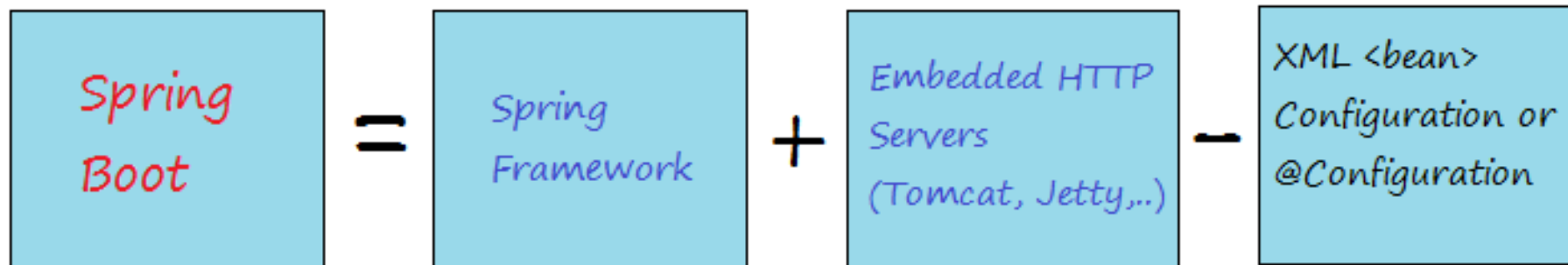


Spring Boot: Installation

Nous allons avant toute chose essayer de bien de nous équiper avant de commencer.

Nous allons télécharger spring tool suite. <https://spring.io/tools>

NB: Il est pas obligatoire, il s'agit juste de gagner en productivité. IL est souvent sous eclipse.



Spring Boot: Installation

Une fois installée vous pouvez le mettre où vous le souhaitez sur votre machine et execute le .exe

Il vous sera alors demandé de choisir votre workspace. Et voilà.

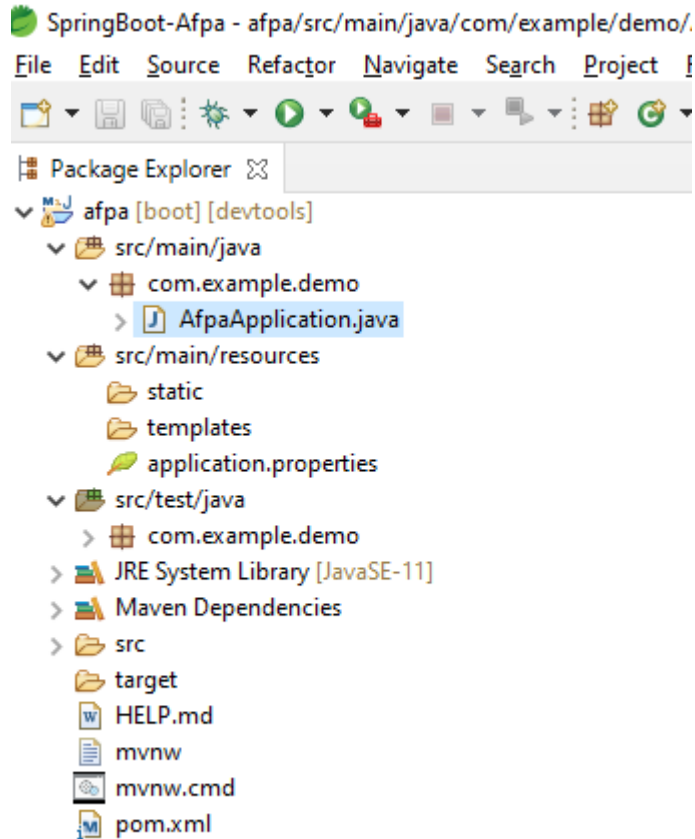


Pour lancer un nouveau projet, nous allons dans File/ New/Spring Starter Projet.

On peut aussi choisir des composants pour notre application:

- Web
- Jpa
- Thymeleaf
- Mysql driver
- Spring dev tools

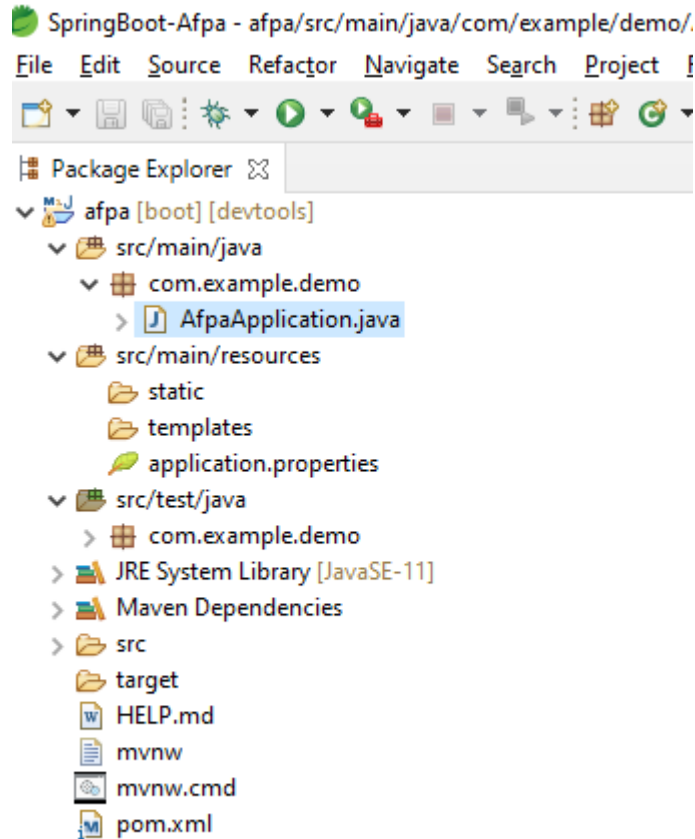
Spring Boot: Architecture



Maven est un outil de gestion et d'automatisation de production des projets logiciels Java en général et Java EE en particulier. Il est utilisé pour automatiser l'intégration continue lors d'un développement de logiciel.

Chaque projet ou sous-projet est configuré par un POM qui contient les informations nécessaires à Maven pour traiter le projet (nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, noms des contributeurs, etc.).

Spring: configuration



Main: contient le code source

Src/main/ressources:

- static: equivalent d'un dossier public, on y met tout ce qui est externe par exemple image, css, js, etc ...

Application.properties: fichier de conf de serveur

Test: pour les tests unitaires

Mvnw / mvnw.cmd:

- signifie maven Wrapper.
- Le .cmd est pour window et les autres pour linux ou mac

Pom: Project Object Model c'est le fichier de configuration de notre projet

Creation de 1er controller

Avec spring Boot, on va en place un modèle MVC.

Comme vous le savez sans doute, le controller est le chef d'orchestra, il joue un role determinant pour le bon fonctionnement de notre application.

Exemple: Créons un controller pour notre page d'accueil

```
package com.example.demo.controller;

import
org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.Ge
tMapping;

@Controller
public class HomeController {
    @GetMapping("/")
    public String Home() {
        return "home";
    }

}
```


Explication

Nous venons de créer le controller avec;

`@Controller` ou `@RestController`

- Cette annotation permet de dire à notre système que nous voulons un controller

`@GetMapping("/")` ou `@RequestMapping("/")`

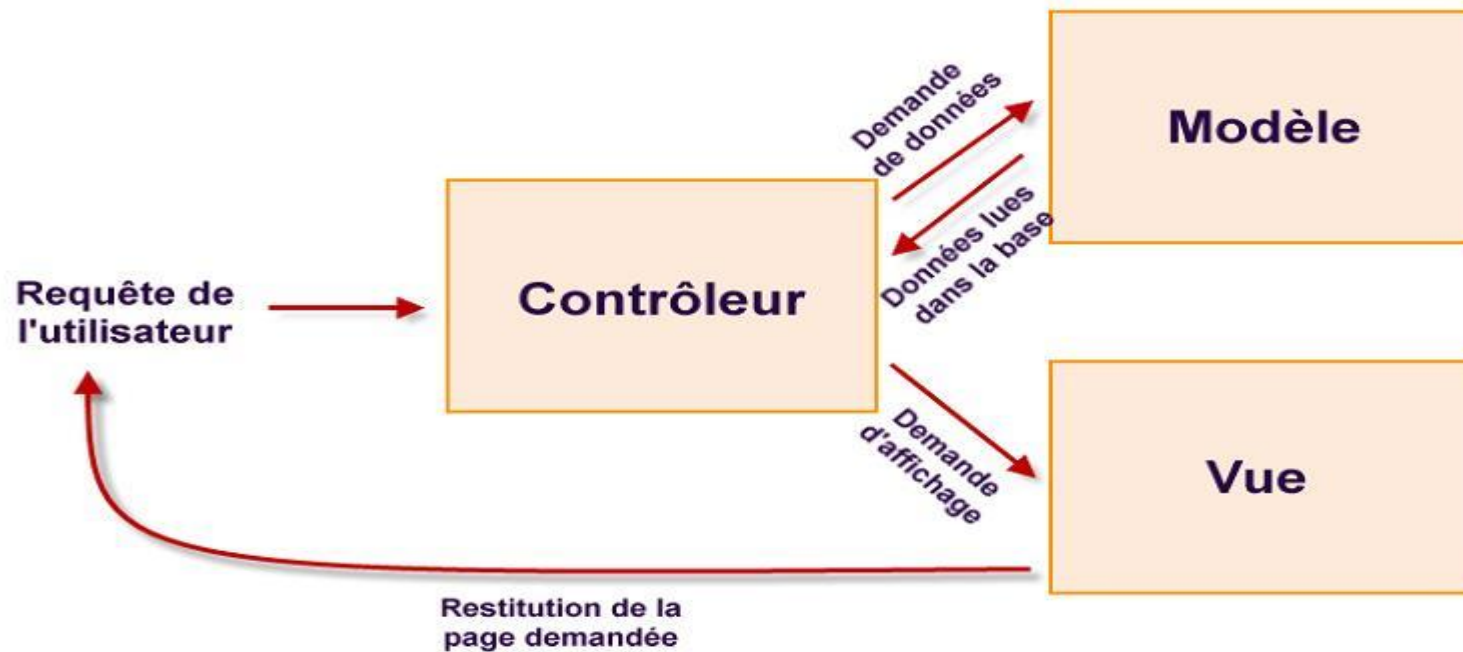
- Définit la route vers la page d'accueil

`return "home";`

- **Fait appel aux templates (Vues) pour afficher la page**

La vue

Partie visible de l'iceberg, elle contient la présentation de l'interface graphique.



La vue: Thymeleaf

Thymeleaf est un moteur de template, sous licence Apache 2.0, écrit en Java pouvant générer du XML/XHTML/HTML5. Thymeleaf peut être utilisé dans un environnement web (utilisant l'API Servlet) ou non web. Son but principal est d'être utilisé dans un environnement web pour la génération de vue pour les applications web basées sur le modèle MVC.

les caractéristiques de Thymeleaf sont les suivantes :

- C'est un moteur de template écrit en Java traitant les fichiers XML, XHTML et HTML5.
- Thymeleaf permet de traiter à la fois les fichiers appartenant à un site web ou non. Il n'y a pas dépendance vis-à-vis de l'API Servlet.



SPRING BOOT



THYMELEAF

Le model

Un modèle (Model) contient les données de traitements.

Le modèle contient les données manipulées par le programme. Il assure la gestion de ces données et garantit leur intégrité. Dans le cas typique d'une base de données, c'est le modèle qui la contient.

Le modèle offre des méthodes pour mettre à jour ces données (insertion suppression, changement de valeur).

Connection à base de donnée

Pour la connection, on peut utiliser les propriétés d'hibernate. Prenons le fichier applications.properties dans Ressources et mettre les accès à notre BDD

- ## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
- `spring.datasource.url = jdbc:mysql://localhost:3306/springBoot-demo?useSSL=false`
- `spring.datasource.username = root`
- `spring.datasource.password =`
- ## Hibernate Properties
- # The SQL dialect makes Hibernate generate better SQL for the chosen database
- `spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect`
- # Hibernate ddl auto (create, create-drop, validate, update)
- `spring.jpa.hibernate.ddl-auto = update`
- `spring.thymeleaf.mode: HTML`

Les services

Créer un package pour avoir toutes les services ainsi les repository

- `package com.example.demo.services;`
- `import org.springframework.beans.factory.annotation.Autowired;`
- `import org.springframework.stereotype.Service;`
- `@Service`
- `public class UserServices {`
- `@Autowired`
- `private UserRepository userRepository;`
- `}`

Les dépendances

S'assurer que nous avons toutes les dépendances dans le pom.xml

Jpa:

```
<dependency>
  ◦ <groupId>org.springframework.boot</groupId>
  ◦ <artifactId>spring-boot-starter-data-
    jpa</artifactId>
</dependency>
```

Mysql:

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-
    java</artifactId>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</grou
    pId>
  <artifactId>spring-boot-starter-
    test</artifactId>
  <scope>test</scope>
</dependency>
```

Exemple

Créer une entité Friend avec les attributs nom, prénom et email



#	Prénom	Nom	Email
1	Moussa	Camara	msa.camara@gmail.com
2	Moussa	Camara	msa.camara@gmail.com

Friend Controller

```
package com.example.demo.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.example.demo.model.Friend;
import com.example.demo.services.FriendServices;

@Controller
public class FriendController {

    @Autowired
    private FriendServices friendServices;

    @GetMapping("/friend")
    public String friend() {
        return "friend/form";
    }

    //***** Ajout user
    @RequestMapping(value = "/list-friend", method = RequestMethod.POST)
    public String add(@Validated Friend friend, Model model) {

        friendServices.createFriend(friend);
        return "redirect:/list";
    }

    @RequestMapping("/list")
    public String list(Model model) {

        List<Friend> friend = friendServices.findAll();

        model.addAttribute("friends", friend);

        return "friend/list";
    }
}
```

Les annotations

L'annotation [@Autowired](#) permet de demander une injection automatique

L'annotation [@Service](#) permet de déclarer un bean de service, c'est à dire de la couche métier, son nom étant passé en paramètre. Cette valeur est d'ailleurs le seul paramètre de cette annotation.

L'annotation [@Scope](#) permet de préciser la portée du bean ; les valeurs possibles pour cette annotation sont "singleton" et "prototype", pour tous les types d'applications, ainsi que "request" et "session" pour les applications Web.

L'annotation [@Repository](#) s'utilise de la même façon, pour les bean DAO (Data Access Object) de la couche de persistance.

Les annotations

@ RequestMapping mappe la méthode du contrôleur avec l'URL fournie dans le formulaire.

@ModelAttribute lie les champs de formulaire à l'objet

Les services

Ensuite nous allons créer les services:

```
import com.example.demo.model.Friend;
import com.example.demo.repository.FriendRepository;

@Service

public class FriendServices {
    @Autowired
    private FriendRepository friendRepository;
    public void createFriend(Friend friend) {
        friendRepository.save(friend);
    }

    //LISTE DES USERS
    public List<Friend> findAll() {
        return friendRepository.findAll();
    }
}
```

Le model

Dans le model (Entité), on a annotation @Entité.

@Entity

```
public class Friend implements Serializable {
```

- @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
- private int id;
- private String nom;
- private String prenom;
- private String Email;

```
package com.example.demo.model;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Friend implements Serializable {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String nom;
    private String prenom;
    private String Email;

    public Friend() {
        super();
        // TODO Auto-generated constructor stub
    }
    public Friend(String nom, String prenom, String email) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        Email = email;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
    public String getPrenom() {
        return prenom;
    }
}
```

Les repository

Repositories sont des interfaces héritant de l'interface Repository. L'objectif de ces interfaces consiste à rendre la création de la couche d'accès aux données (requêtes SELECT, UPDATE...) plus rapide.

On n'est donc pas obligés d'écrire des requêtes supplémentaires pour retrouver une entité, par exemple, par son identifiant ou de retrouver toutes les entités disponibles. Les méthodes findById() et findAll() sont là pour cela. En plus, on peut utiliser les méthodes liées au CRUD (Create, Read, Update, Delete) sans aucun effort supplémentaire.

```
import org.springframework.data.jpa.repository.JpaRepository;

import com.example.demo.model.Friend;

public interface FriendRepository extends JpaRepository<Friend, Long> {
    Friend findOneById(Long id);
}
```

Les repositories - repository

L'EntityRepository propose par défaut quelques méthodes pour vous éviter de les écrire.

- **find** prend un unique paramètre et recherche l'argument dans la clé primaire de l'entité.
- **findBy** prend 4 paramètres (\$criteria, \$orderBy, \$limit, \$offset). Cette méthode retourne des résultats correspondant aux valeurs des clés demandées.
- **findAll** est un alias de findBy([]). Il retourne par conséquent tous les résultats.
- **findOneBy** fonctionne comme la méthode findBy mais retourne un unique résultat et non pas un tableau

Et la vue

On va créer le form.html et list:

```
<div class="container">
  <form method="post" th:action="@{/list}" th:object="${Friend}">
    <div class="form-row">
      <div class="form-group col-md-6">
        <label for="versement">Nom </Label>
        <input type="text" class="form-control" id="nom" name="nom" required />
      </div>
    </div>

    <div class="form-row">
      <div class="form-group col-md-6">
        <label for="versement">Prénom</Label>
        <input type="text" class="form-control" id="prenom" name="prenom" required />
      </div>
    </div>

    <div class="form-row">
      <div class="form-group col-md-6">
        <label for="versement">Email</Label>
        <input type="email" class="form-control" id="email" name="email" required />
      </div>
    </div>

    <input type="submit" value="Ajouter" class="btn btn-primary" />
  </form>
</div>
```

*th:action sert à spécifier
l'url où sera soumis le
formulaire*

*th:object sert à spécifier
l'objet où sera lié les
données soumises au
formulaire*

Review



Revoyons ce qu'on a fait:

- Installé Spring Boot avec suite tools
- On a généré l'architecture MVC
- Ensuite on a généré une entité (un model).
- Créer une base de donnée
- Créer l'entité User avec nom, prénom

Avec ces quelques lignes de commandes nous avons une application fonctionnelle. Il nous suffit juste de faire un peu de personnalisation et d'envoyer en prod. 😊

Thymeleaf

Thymeleaf est un Java XML/XHTML/HTML5 Template Engine qui peut travailler à la fois dans des environnements Web (Servlet) et celui de non Web. Il est mieux adapté pour diffuser XHTML/HTML5 sur View (View Layer) des applications Web basées sur MVC. Mais il peut traiter n'importe quel fichier XML même dans des environnements hors ligne (offline). Il fournit une intégration complète de Spring Framework.

Le fichier modèle (Template file) de Thymeleaf est en substance un fichier de document ordinaire au format XML/XHTML/HTML5. Thymeleaf Engine (le moteur Thymeleaf) va lire un fichier modèle et le combiner avec des objets Java pour générer (generate) un autre document.

Model

```
public class Person {  
    private String firstName;  
    private String lastName;  
}
```

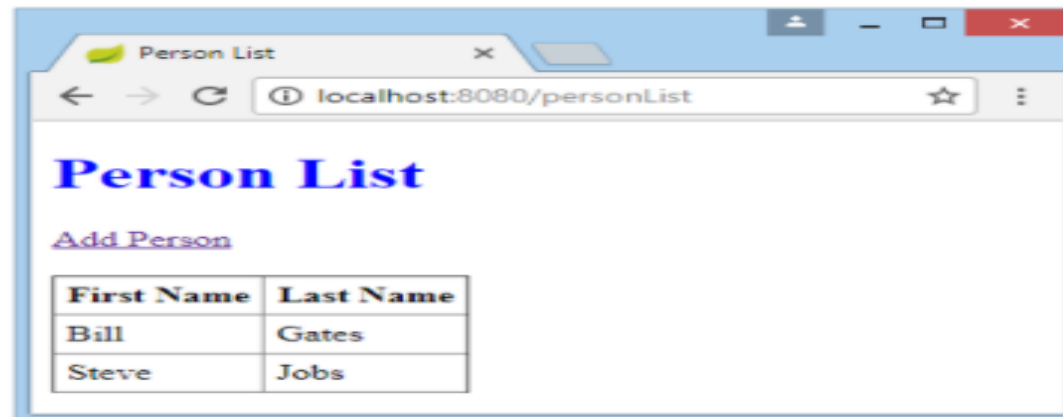
List<Person> persons



View (Thymeleaf Template)

```
<!DOCTYPE HTML>  
<html xmlns:th="http://www.thymeleaf.org">  
    <head>  
        <meta charset="UTF-8" />  
        <title>Person List</title>  
        <link rel="stylesheet" type="text/css"  
              th:href="@{/css/style.css}"/>  
    </head>  
    <body>  
        <h1>Person List</h1>  
        <a href="addPerson">Add Person</a>  
        <br/><br/>  
        <div>  
            <table border="1">  
                <tr>  
                    <th>First Name</th>  
                    <th>Last Name</th>  
                </tr>  
                <tr th:each="person : ${persons}">  
                    <td th:utext="${person.firstName} ">...</td>  
                    <td th:utext="${person.lastName} ">...</td>  
                </tr>  
            </table>  
        </div>  
    </body>  
</html>
```

Thymeleaf Engine



Thymeleaf

Thymeleaf Template est un fichier modèle. Son contenu est au format XML/XHTML/HTML5.

Tous les fichiers HTML doivent déclarer l'utilisation de Thymeleaf Namespace:

```
<!-- Thymeleaf Namespace -->
```

```
<html xmlns:th="http://www.thymeleaf.org">
```

Tous les fichiers sont dans Ressources/templates de notre application.

Quelques exemples

- `th:text="${text}"` : Ecrit au sein du tag la variable `text` du modèle
 - ``
- `th:if="${boolean}"` : Affiche le tag courant que si la condition est vérifiée
 - ``
- `th:unless="${boolean}"` : Affiche le tag courant que si la condition n'est pas vérifiée
 - ``
- `th:href="@{/path}"` : Remplace le tag `href` courant par la valeur de l'url (dans une balise `a`)
 - `<a th:href="@{/mypath/abc.html}">A Link`
- `th:action="@{/path}"` : Idem que la balise `th:href` pour un formulaire
 - `<form action="#" th:action="@{/saveStudent}" th:object="${student}" method="post">`
- `th:each="item : ${items}"` : Itère sur un tableau, une liste, une collection ou un iterator
 - `<tr th:each="student: ${students}"> <td th:text="${student.name}" /></tr>`

Thymeleaf

- <https://www.thymeleaf.org/documentation.html>



+



Exercice - Employés

Faire un programme permettant de ~~créer~~ et d'afficher la liste des employés

~~Chaque employé a un nom, prénom, email et fonction~~

Les étapes:

- ~~Créer l'entité~~
- ~~Créer le controller~~
- ~~Créer la repositorie~~
- Et les vues
- FAIRE LA PARTIE UPDATE ET DELETE DES EMPLOYES

Spring Boot: Blog



Spring Boot: Blog

- Pour notre blog, nous allons avoir besoin d'un système de gestion des utilisateurs.
- Les utilisateurs non identifiés pourront lire les articles (Post)
- Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
- Les utilisateurs identifiés pourront commenter des articles (Post)



TP – Cahier des charges

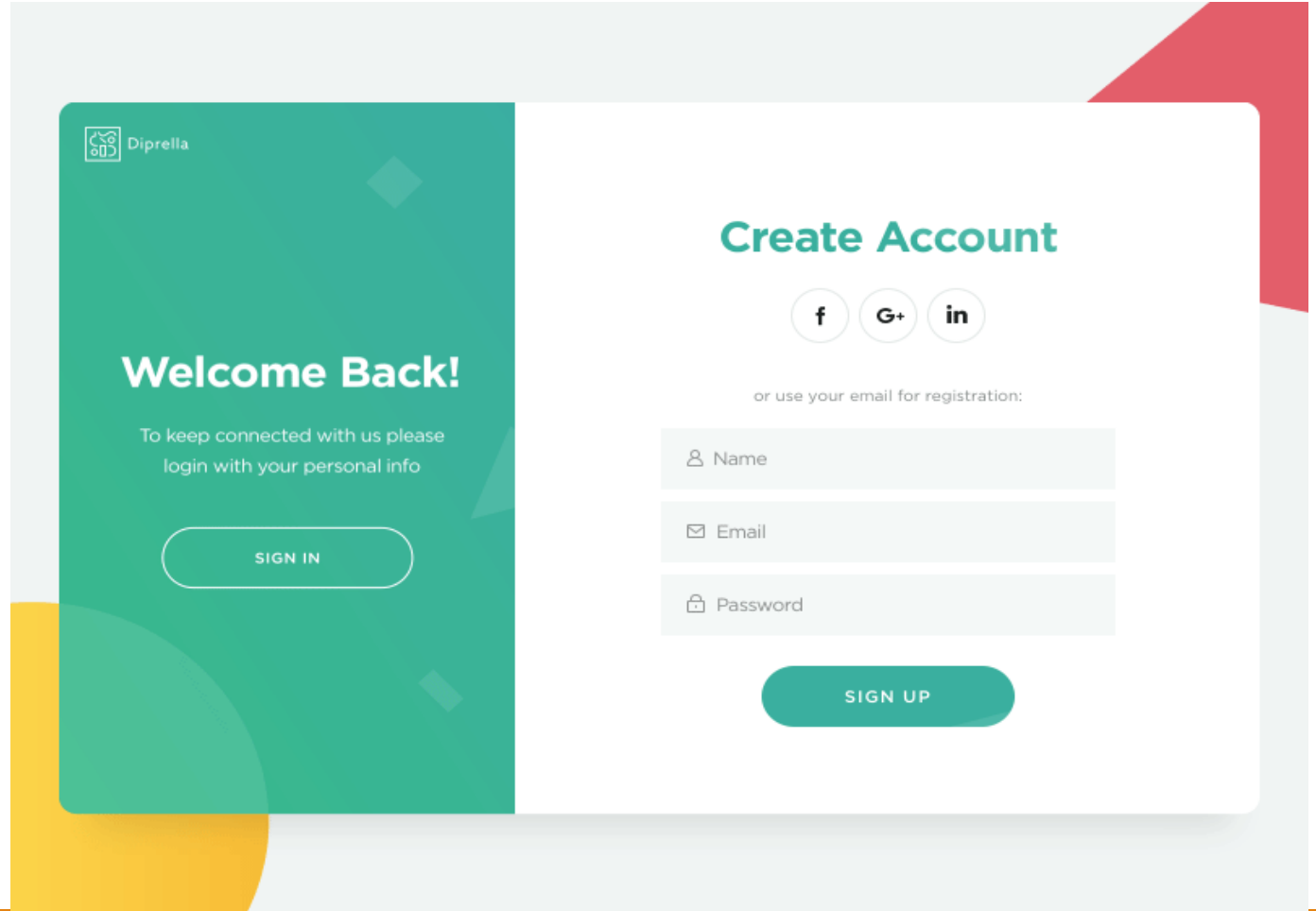
- Reprenons notre cahier pour refaire l'application avec Spring boot
- Pour notre blog, nous allons avoir besoin d'un **système de gestion des utilisateurs**.
- Les utilisateurs non identifiés pourront lire les articles (Post)
- Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
- Les utilisateurs identifiés pourront commenter des articles (Post)



Sign in / Sign up

Système de gestion des utilisateurs.

Chaque utilisateur doit pouvoir se connecter à notre blog mais pour cela, il faut déjà avant créer son compte



The image displays a user management interface with two main sections: a 'Welcome Back!' login area and a 'Create Account' registration area.

Welcome Back! Section:

- Logo: Diprella
- Text: **Welcome Back!**
- Text: To keep connected with us please login with your personal info
- Button: SIGN IN

Create Account Section:

- Text: **Create Account**
- Social Login: Buttons for Facebook (f), Google+ (G+), and LinkedIn (in)
- Text: or use your email for registration:
- Form Fields: Name, Email, Password
- Button: SIGN UP

Register form

Nous allons créer un formulaire de login et register pour les utilisateurs

Commençons par le register:

- Création entité User avec nom, prenom, email et password

Entité User

Cela donne :

- @Entity
- **public class** User {
- @Id @GeneratedValue(strategy = GenerationType.*IDENTITY*)
- **private int** id;
- **private String** nom;
- **private String** prenom;
- **private String** email;
- **private String** password;
- NB: NE PAS OUBLIER CONSTRUCTEUR, GETTERS ET SETTERS

Spring boot: register - Controller

```
@Controller
public class RegisterController {

    @Autowired
    private UserServices userServices;

    @RequestMapping(value = "/register", method =
RequestMethod.GET)
    public String register(Model model){
        model.addAttribute("user", new User());

        return ("user/register");
    }
}
```

```
@PostMapping("/register")
    public String register(@Validated User user,
BindingResult bindingResult, Model model) {

        model.addAttribute("user", new User());
        if(bindingResult.hasErrors()) {
            return ("user/register");
        }

        userServices.createUser(user);
        return ("user/success");
    }
}
```

Spring boot - Service

Créons le service:

Rappel: service est une classe
qui réalise des traitements
métiers sur des données.

```
@Service
public class UserServices {

    @Autowired
    private UserRepository userRepository;

    public void createUser(User user) {
        BCryptPasswordEncoder encoder = new
BCryptPasswordEncoder();

        user.setPassword(encoder.encode(user.getPassword()));

        userRepository.save(user);
    }
}
```

Spring boot – la vue

Créons maintenant la vue - thymeleaf

```
<form method="post" th:action="@{/register}" th:object="${user}">

    <div class="form-group">
        <label for="firstname" class="form-control-label">Nom</label>
        <input type="text" class="form-control" id="nom" name="nom" required />
    </div>

    <div class="form-group">
        <label for="username" class="form-control-label">Prénom</label>
        <input type="text" class="form-control" id="prenom" name="prenom" required />
    </div>

    <div class="form-group">
        <label for="email" class="form-control-label">Email</label>
        <input type="text" class="form-control" id="email" name="email" required />
    </div>

    <div class="form-group">
        <label for="password" class="form-control-label">Password</label> <input
            type="password" class="form-control" name="password" required />
    </div>

    <input type="submit" value="Je crée mon compte" class="btn btn-primary" />

</form>
```


Spring boot - login

Après la création de compte, on va maintenant faire UserLogin qui implémente UserDetails. cette classe implémente interface UserDetails.

L'interface fournit des informations sur l'utilisateur principal. Les implémentations ne sont pas utilisées directement par Spring Security à des fins de sécurité.

```
public class UserLogin implements UserDetails {  
  
    private User user;  
  
    public UserLogin(User user) {  
        this.user = user;  
    }  
  
    @Override  
    public Collection<? extends  
GrantedAuthority> getAuthorities() {  
        return null;  
    }  
  
    @Override  
    public String getPassword() {  
        return user.getPassword();  
    }  
}
```

Spring boot - UserLoginDetail

```
public class UserLoginDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepo;

    @Override
    public UserDetails loadUserByUsername(String username) throws
        UsernameNotFoundException {
        User user = userRepo.findByEmail(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not found");
        }
        return new UserLogin(user);
    }
}
```

Spring boot - security

La configuration de Spring Security en Java config est relativement simple et directe. Elle permet de contrôler et de centraliser les accès à toutes les URL de l'application. De plus, il est possible de définir très simplement les URL de login et de logout, Spring Security se chargeant des traitements des requêtes.

Spring Security est un Framework de sécurité léger qui fournit une authentification et un support d'autorisation afin de sécuriser les applications Spring. Il est livré avec des implémentations d'algorithmes de sécurité populaires.

Commençons par une secu basique de spring, rajoutons dans notre pom.xml

```
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-security</artifactId>  
</dependency>
```

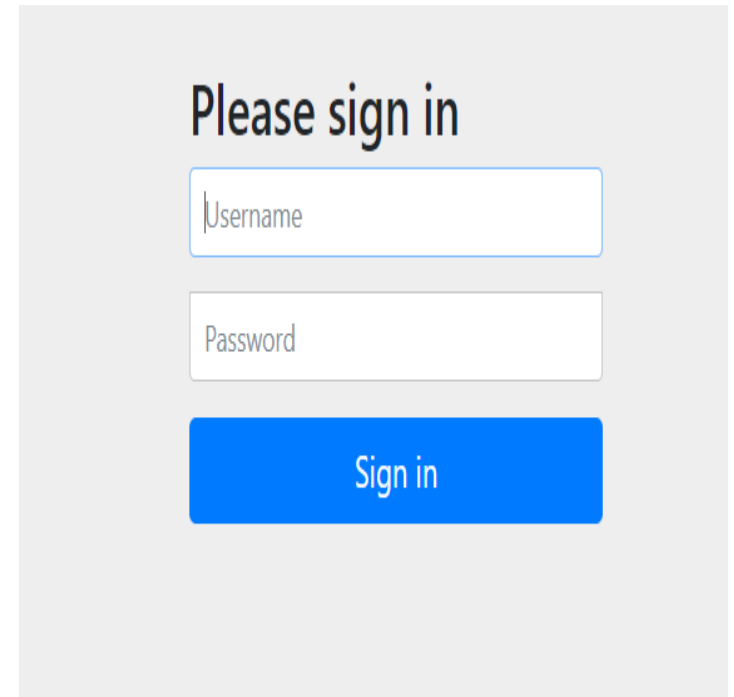
En rajoutant cette sécurité, nous remarquons maintenant qu'à chaque fois que nous essayons d'accéder à notre application, nous avons un popup nous demandant un login et mot de passe.

Oups ... !

Faisons un peu de config dans le fichier application.properties pour rajouter un user et un mot de passe:

```
spring.security.user.name=admin
```

```
spring.security.user.password=1234
```

A screenshot of a web application's login page. The page has a light gray background. At the top, the text "Please sign in" is displayed in a bold, black font. Below this, there are two input fields: the first is labeled "Username" and the second is labeled "Password". Both fields are white with a thin blue border. Below the password field is a blue button with the text "Sign in" in white. The entire form is centered on the page.

Spring peut nous laisser le droit de gerer notre propre securité comme on l'attend.

Pour cela, nous allons une classe `SecurityConfig.java` qui herite de `WebSecurityConfigurerAdapter` de Spring boot

configure (AuthenticationManagerBuilder) est utilisé pour établir un mécanisme d'authentification en permettant aux AuthenticationProviders d'être ajoutés facilement: par exemple, ce qui suit définit l'authentification en mémoire avec les connexions intégrées «utilisateur» et «admin».

```
@Override
```

```
    protected void configure(AuthenticationManagerBuilder auth) throws  
Exception {  
        auth.authenticationProvider(authenticationProvider());  
    }
```

configure (HttpSecurity) permet la configuration de la sécurité basée sur le Web au niveau des ressources, en fonction d'une correspondance de sélection

par exemple, l'exemple ci-dessous limite les URL commençant par / admin / aux utilisateurs qui ont le rôle ADMIN, et déclare que toutes les autres URL doivent être authentifié avec succès.

```
@Override
    protected void configure(HttpSecurity http)
    throws Exception {
        http.authorizeRequests()
            .antMatchers("/users").authenticated()
            .anyRequest().permitAll()
            .and()
            .formLogin()
            //page de login créée
            .loginPage("/login")
            .usernameParameter("email")
                .defaultSuccessUrl("/list")
                .permitAll()
            .and()

            .logout().logoutSuccessUrl("/").permitAll();
    }
```

configure (WebSecurity) est utilisé pour les paramètres de configuration qui ont un impact sur la sécurité globale (ignorer les ressources, définir le mode de débogage, rejeter les demandes en implémentant une définition de pare-feu personnalisée).

Par exemple, la méthode suivante ferait en sorte que toute demande commençant par /resources/ soit ignorée à des fins d'authentification.

```
@Override
    public void configure(WebSecurity web)
        throws Exception {

        web.ignoring().antMatchers("/resources/**",
            "/static/**", "/css/**", "/js/**",
            "/images/**");
    }
```


Review: gestion des utilisateurs

1. Création de compte:
 1. Package: RegisterController avec Get et Post Mapping
2. Service
 1. UserServices
 1. `createUser(User user)`
3. Repository
4. Création de la vue avec thymeleaf



Resumons

Service

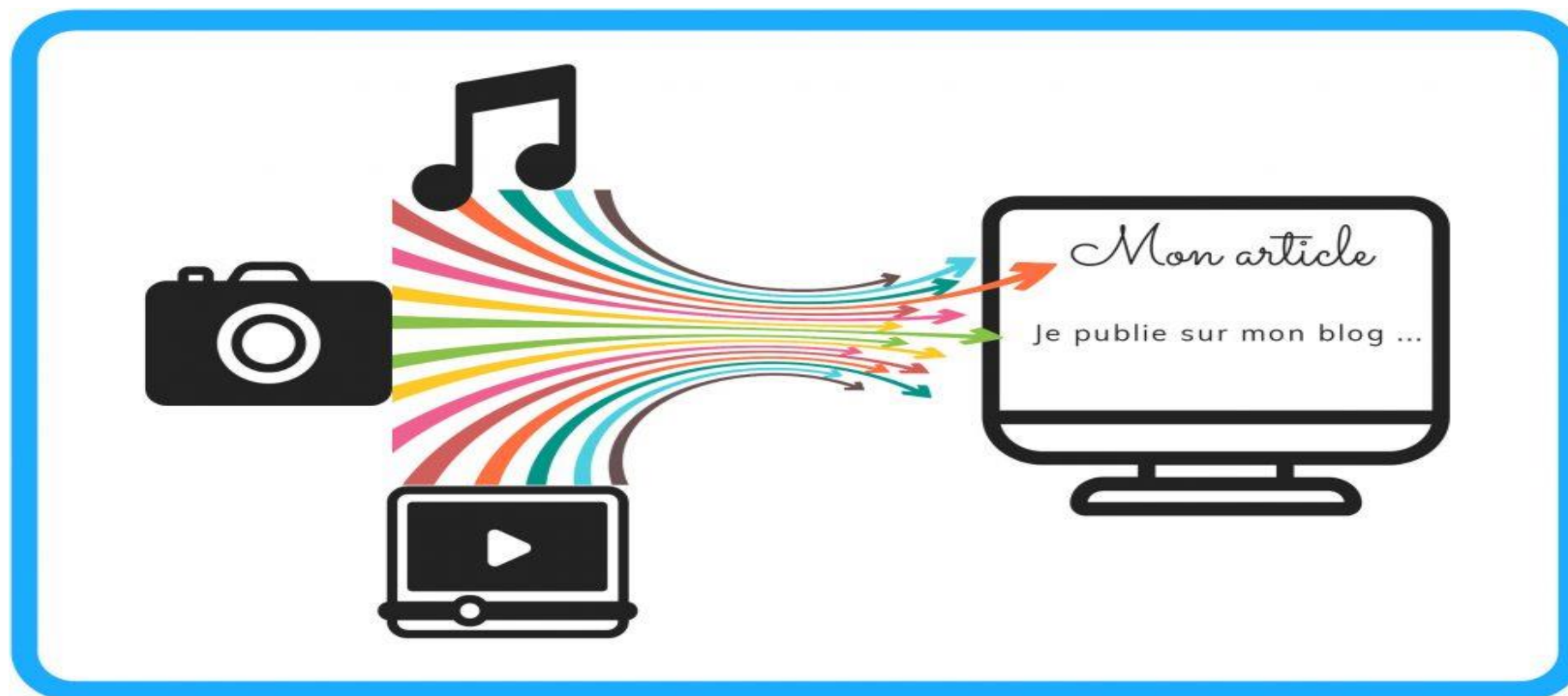
Autowired

Spring boot -

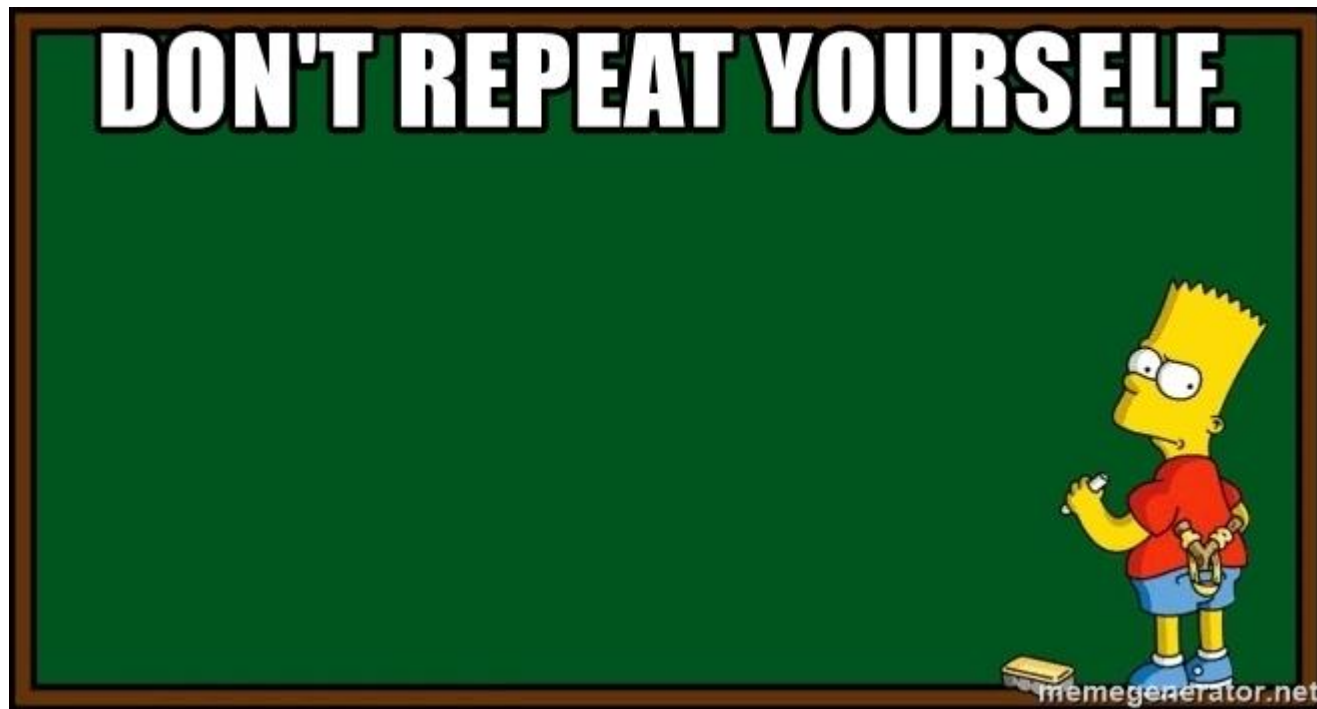
- Reprenons notre cahier pour refaire l'application avec Spring boot
- ~~◦ Pour notre blog, nous allons avoir besoin d'un système de gestion des utilisateurs.~~
- Les utilisateurs non identifiés pourront lire les articles (Post)
- Les utilisateurs identifiés et autorisés pourront poster des articles (Post)
- Les utilisateurs identifiés pourront commenter des articles (Post)



Commençons par le model, le controller sans oublier le service



Pour la vue, nous allons sans doute créer un fichier html comme pour le User avec un peu de bootstrap. Nous allons certainement dupliqué du code (Partie header et footer)



DRY signifie «Don't Repeat Yourself», un principe de base du développement logiciel visant à réduire la répétition de code.

Le principe DRY est énoncé comme suit:
« Chaque élément doit avoir une représentation unique et non ambiguë dans un système. »

Avec Thymeleaf, nous avons la possibilité d'utiliser des fragments.

Un Fragment fait partie d'un Template. Thymeleaf vous permet d'importer des fragments de ce Template dans un autre Template.

Il existe de nombreuses façons d'identifier un Fragment.

- Sélectionnez toutes les étiquettes (tag) avec l'attribut (attribute) `th:fragment="search-fragment"`.
- Sélectionnez l'étiquette par ID.
- Sélectionnez toutes les étiquettes par Css-class.

L'important, lorsque vous voulez importer un Fragment d'un Template, c'est que vous devez décrire **sa position**.

La syntaxe pour décrire la position d'un Fragment d'un Template :

- `~/path-to-template/template-name :: selector`

Reprenons notre header et footer.

Header.html

FRAGMENT/HEADER.HTML

```
<header th:fragment="header"
xmlns:th="http://www.w3.org/1999/xhtml"
xmlns:sec="http://www.w3.org/1999/xhtml">
  <div class="card-header">
    <ul class="nav nav-tabs card-header-tabs">
      <li class="nav-item">
        <a href="index.html">Accueil</a>
      </li>
    </ul>
  </div>
</header>
```

DANS HOME.HTML

```
</head>
<body>
  <!-- Appel fragment header -->
  <div th:replace="fragments/header::header">
</div>

  <div class="container">

    <h2>Qu'est-ce que le Lorem Ipsum?</h2>
```

Pour le moment, tous les utilisateurs peuvent poster des articles or dans notre cahier des charges, seul les utilisateurs identifié et autorisé devraient pouvoir poster des articles

Commençons par l'authentification. On s'occupera de l'autorisation plus tard

Les utilisateurs identifiés et autorisés pourront poster des articles (Post)

Rajouter dans notre securityConfig

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/ajout-article").authenticated()  
}
```

Il va falloir pouvoir dire à notre application que si:

- Un utilisateur n'est pas connecté, il ne doit pas pouvoir accéder au formulaire de création d'un Post.
- Un utilisateur est connecté, lorsqu'il créer un Post, le Post doit lui être associé

Cela implique que un Post est forcément lié à un utilisateur

Un utilisateur devrait pouvoir poster plusieurs Posts.



Comment faire ?

Spring boot – les relations

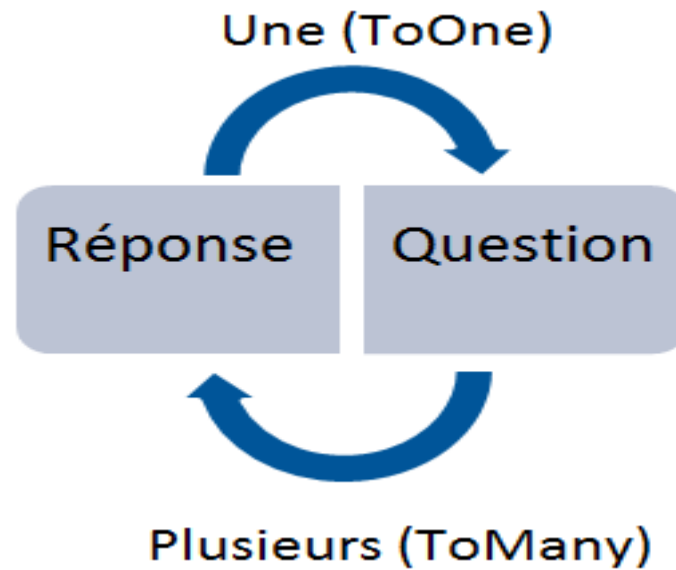
Une des fonctionnalités majeures des ORM est de gérer les relations entre objets comme des relations entre tables dans un modèle de base de données relationnelle. JPA définit des modèles de relation qui peuvent être déclarés par annotation.

On peut définir quatre types de relations entre entités JPA :

- relation 1:1 : annotée par `@OneToOne` ;
- relation n:1 : annotée par `@ManyToOne` ;
- relation 1:p : annotée par `@OneToMany` ;
- relation n:p : annotée par `@ManyToMany`.

Dans notre cas, un utilisateur peut avoir plusieurs posts et un post ne peut avoir qu'un utilisateur.

C'est donc un **Many-To-One** dans notre entité Post.



Reprenons notre entité Article et rajoutons cette relation.

Deux types de relations s'offrent à nous:

Une relation **unidirectionnelle**

Une relation **bidirectionnelle**

Spring boot - relation **unidirectionnelle**

```
@Entity
public class Article {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String titre;
    private String resume;
    private String contenu;
    private Date created_at;

    @ManyToOne
    private User user;
```

Ici, nous avons une seule relation dans la classe Article.

JPA a créé une colonne de jointure dans la table Article, et une clé primaire qui référence la clé primaire de la table User.

En effet, JPA essaie de rajouter user_id grace à la nouvelle relation et constraint

PS: mettre à jour la base de donnée en rajoutant une colonne user_id dans la table Article

Maintenant on va créer un nouveau Post et on va aller voir ce qu'il y a dans notre base de données comment ça s'est rempli...

La colonne user_id de post est NULL !!!

Pour un tel ajout, vous devez

- 'Setter' l'utilisateur explicitement depuis votre controleur

Pour pouvoir attribuer un utilisateur à notre propriété user qui je vous le rappelle est privé, nous devons utiliser un setter.

Nous allons aussi en profiter pour rajouter un getter

Et maintenant nous n'avons plus qu'à utiliser notre setter depuis notre contrôleur pour ajouter l'utilisateur courant à notre Post comme ceci:

```
//On recupere le nom de l'utilisateur connecté
```

```
String userName = ((UserLogin) SecurityContextHolder.getContext().getAuthentication().getPrincipal()).getUsername();
```

```
//Instancie un nouvel objet avec le mail de user connecté
```

```
User user = new User();
```

```
user = userRepository.findByEmail(userName);
```

```
article.setUser(user);
```

```
articleService ajoutArticle(article);
```

Désormais lorsqu'on créer un nouveau post un utilisateur lui est associé.

Si on se déconnecte et qu'on créer un nouveau post, la colonne user de Post sera à null.

Corrigeons cette partie.



Spring boot - security

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/users", "/ajout-article").authenticated()
        .anyRequest().permitAll()
        .and()
        .formLogin()
        //page de login créée
        .loginPage("/login") .usernameParameter("email")
            .defaultSuccessUrl("/list")
            .and()
            .logout()
            .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
            .logoutSuccessUrl("/")
            .permitAll()
        .and()
        .logout().logoutSuccessUrl("/").permitAll()
    ;
}
```

Spring boot - relation **bidirectionnelle**

Une relation bidirectionnelle doit correspondre à une relation p:1 dans la classe destination de la relation. Comme pour le cas des relations 1:1, le caractère bidirectionnel d'une relation 1:p est marqué en définissant l'attribut mappedBy sur la relation.

Attention: JPA nous pose une contrainte ici : l'attribut mappedBy est défini pour l'annotation @OneToMany, mais pas pour l'annotation @ManyToOne.

Exemple – relation bidirectionnelle

USER.JAVA

```
@Id @GeneratedValue(strategy =
 GenerationType.IDENTITY)

    private Long id;
    private String nom;
    private String prenom;
    private String email;
    private String password;

    @OneToMany(mappedBy="user")
    private Collection<Article> article ;
```

ARTICLE.JAVA

```
@Id @GeneratedValue(strategy =
 GenerationType.IDENTITY)

    private Long id;
    private String titre;
    private String resume;
    private String contenu;
    private Date created_at;

    @ManyToOne
    private User user;
```

Relation: ManyToMany

Reprenons notre blog et rajoutons maintenant des roles à nos utilisateurs. Supposons que plusieurs roles pour plusieurs utilisateurs.

Cela donne une relation ManyToMany entre les tables user et role:

Dans la classe:

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(name = "user_roles",
    joinColumns={ @JoinColumn(name = "user") },
    inverseJoinColumns = { @JoinColumn(name = "role") })
private List<Role> roles;
```

Les cascades

Avec JPA, il est possible de propager des modifications à tout ou partie des entités liées. Les annotations permettant de spécifier une relation possèdent un attribut cascade permettant de spécifier les opérations concernées : **ALL, DETACH, MERGE, PERSIST, REFRESH ou REMOVE**.


Entity Bean Compliments

- **Fetch**: option for loading the graph of objects
 - **FetchType.EAGER** : loads all the tree (required if Serializable)
 - **FetchType.LAZY** : only on demand (unusable with Serializable)
- **Cascade**: transitivity of operations over the beans
 - **CascadeType.ALL**: every operation is propagated
 - **CascadeType.MERGE**: in case of a merge
 - **CascadeType.PERSIST**: When film Entity becomes persistent, then List<Cinema> too
 - **CascadeType.REFRESH**: loading from the DB
 - **CascadeType.REMOVE**: delete in cascade

42

Fahad R. Gobra

JEE - Java Persistence, JPA 2.x



Relation: ManyToMany

Créons notre table role avec ces attributs

```
@Entity
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "roles")
    private List<User> users;
```

Ici, nous allons mapper le role aux users

Entités

@Entity

public class Role {

 @Id

 @GeneratedValue(strategy = GenerationType.*IDENTITY*)

private Long *id*;

private String *name*;

 @ManyToMany(mappedBy = "*roles*")

private List<User> *users*;

public Role(String name, List<User> users) {
 super();
 this.name = name;
 this.users = users;
 }

UserController

Nous allons maintenant mettre à jour notre userController:

```
@RequestMapping(value = "/addUser", method = RequestMethod.GET)

    public String register(Model model){

        model.addAttribute("user", new User());
        model.addAttribute("roles", roleServices.findAll());
        return ("user/ajout");

    }

@RequestMapping(value = "/addUser", method = RequestMethod.POST)

    public String register(@Validated User user, BindingResult bindingResult, Model model) {

        if(bindingResult.hasErrors()) {

            return ("user/ajout");

        }

        userServices.createUser(user);
        return "redirect:/listUser";

    }
```

Et maintenant ... la vue

```
<tbody>
  <tr th:each="user:${users}">
    <td th:text="${user.id}"> </td>
    <td th:text="${user.prenom}"> </td>
    <td th:text="${user.nom}"> </td>
    <td th:text="${user.email}"> </td>
    <td th:each="nomRole : ${user.roles}" th:text="${nomRole}"></td>
  </tr>
</tbody>
```

Relation oneToOne

Prenons comme exemple un modèle simple comportant un pays et un président.

Chaque pays a un et un seul président et un président ne peut présider qu'un et un seul pays.

Nous avons ainsi un OneToOne

```
@Entity
public class Pays implements
Serializable {

    @Id
    @GeneratedValue(strategy =
GenerationType.AUTO)
    private Long id;

    @Column(length=40)
    private String nom ;

    @OneToOne
    private President president ;

    // suite de la classe
}
```

Relation OneToMany

Une relation OneToMany est caractérisée par un champ de type Collection dans la classe maître.

Cette annotation ne peut être utilisée qu'avec une collection d'éléments puisqu'elle implique qu'il peut y avoir plusieurs entités associées.

Prenons comme exemple un modèle comportant un projet et une équipe de développeurs.

Un projet a plusieurs développeurs: OneToMany

En cas de bidirectionnelle

Dans l'entité projet, on aura:

```
@OneToMany  
private Collection<Developeurs> developpeurs ;
```

Dans l'entité developpeurs, on aura:

```
@ManyToOne  
private Projet projet ;
```

Mini projet: vtc



Spring boot - Correction VTC

LA
CORRECTION.